Adaptive Data Parallelism for Internet Clients on Heterogeneous Platforms

Alessandro Pignotti
Scuola Superiore Sant'Anna, Italy
a.pignotti@sssup.it

Adam Welc
Oracle Labs, USA
adam.welc@oracle.com

Bernd Mathiske
Adobe, USA
bernd.mathiske@adobe.com

Abstract

Today's Internet is long past static web pages filled with HTML-formatted text sprinkled with an occasional image or animation. We have entered an era of Rich Internet Applications executed locally on Internet clients such as web browsers: games, physics engines, image rendering, photo editing, etc. Yet today's languages used to program Internet clients have limited ability to tap to the computational capabilities of the underlying, often heterogeneous, platforms.

In this paper we present how a Domain Specific Language (DSL) can be integrated into ActionScript, one of the most popular scripting languages used to program Internet clients and a close cousin of JavaScript. We demonstrate how our DSL, called ASDP (ActionScript Data Parallel), can be used to enable data parallelism for existing sequential programs. We also present a prototype of a system where data parallel workloads can be executed on either CPU or a GPU, with the runtime system transparently selecting the best processing unit, depending on the type of workload as well as the architecture and current load of the execution platform. We evaluate performance of our system on a variety of benchmarks, representing different types of workloads: physics, image processing, scientific computing and cryptography.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features

General Terms languages, performance

Keywords programming languages, data-parallelism, ActionScript

1. Introduction

Today we are in the midst of the multi-core era, but evolution of computational platforms did not stop with the introduction of multi-core. Processing units that have been previously only used for graphics processing, namely GPUs, are rapidly evolving towards supporting general-purpose computations. Consequently, even a modestly priced and equipped modern machine constitutes a heterogeneous computational platform where execution of what has been traditionally viewed as CPU workloads can be potentially delegated to a GPU.

At the same time, today's Internet clients, such as various web browsers, despite growing demand for computational capabilities triggered by the rise of Rich Internet Applications, have very limited ability to utilize computational power available on modern heterogeneous architectures. Attempts have been made to enable utilization of multiple cores, via task parallelism [10] or data parallelism [13], but our system is the first solution in this context where the same data parallel workload can be transparently scheduled on *either* a CPU or a GPU, depending on the type of workload as well as on the architecture and the current load of the underlying platform. We introduce a Domain Specific Language (DSL) [1], that integrates seamlessly with ActionScript, one of the most popular languages used to program Internet clients. Our DSL, called ASDP (ActionScript Data Parallel) is compiled to OpenCL, enabling execution on a GPU or CPU on platforms that support execution of OpenCL programs on these processing units. ASDP is also compiled back to ActionScript to provide a baseline execution mode and to support execution of data parallel programs on platforms that do not support OpenCL. While our solution is based on ActionScript, it can be adapted to other languages used to program Internet clients, such as JavaScript, as it does not rely on any language features that are specific only to ActionScript.

In summary, this paper makes the following contributions:

- We present a domain specific language, ASDP (ActionScript Data Parallel) that closely resembles ActionScript and that can be used to express data parallel computations in the context of ActionScript. We also discuss ActionScript extensions required to integrate ASDP code.
- We describe the implementation of a prototype system responsible for compiling and executing ASDP code fragments. A modified version of the upcoming ActionScript Falcon compiler [19] translates ASDP code fragments to OpenCL (for efficient execution on both a CPU and a GPU) and back to ActionScript (baseline execution mode). A modified version of Tamarin, an open source ActionScript virtual machine, profiles execution of the ASDP code fragments and chooses the best processing unit to execute a given fragment, depending on the type of workload as well as the architecture and current load of the underlying platform.
- We present a performance evaluation of our system and demonstrate its ability to both achieve execution times that are close to those of the best processing unit on a given platform, and to modify processing unit selection whenever the load of the execution platform changes.

[1] Our language is a DSL in a similar sense to, say, OpenGL[8] – its expressiveness is limited and it is focused on a specific domain, that is data parallel programming.

```
for (var i:int = 1; i < array_rows; i++) {
  TestArray[0][i] = TrapezoidIntegrate(0.0f, 2.0f, 1000, omega * i, 1);
  TestArray[1][i] = TrapezoidIntegrate(0.0f, 2.0f, 1000, omega * i, 2);
}
```

Figure 1: Series sequential – main loop

```
TestArray[0] = .TrapezoidIntegrateDP[array_rows, TestArray[0]]
  (0.0f, 2.0f, 1000, omega, 1, TestArray[0][0]);

TestArray[1] = .TrapezoidIntegrateDP[array_rows, TestArray[1]]
  (0.0f, 2.0f, 1000, omega, 2, TestArray[1][0]);
```

Figure 2: Series data parallel – parallel call

```
function .TrapezoidIntegrateDP(x0:float, x1:float, nsteps:int, omega:float,
                               select:int, firstElement:float):float {
  if (index == 0) return firstElement;
  var omegan:float = omega * index;
  return TrapezoidIntegrate(x0, x1, nsteps, omegan, select);
}
```

Figure 3: Series data parallel – kernel definition

2. Motivation

Data parallelism is one of the more natural ways of introducing parallelism in the context of scripting languages, such as ActionScript, as support for data parallelism can be restricted to automatically prevent problems related to managing access to shared memory, such as data races, resulting in safer and more robust programs, which preserves the spirit of scripting languages used to program Internet clients. It is also often easy to parallelize sequential loops using data parallel constructs, with little modifications to the existing code. One of our main goals when developing our solution was to make it convenient to use by "scripters" and yet powerful enough to deliver significant performance gains at the cost of relatively little programming effort. We argue the first point, the ease of use, below by describing modifications required to parallelize one of the benchmarks used for our performance evaluation [2]. We argue the second point, performance, in Section 5.

We demonstrate how a sequential program can be parallelized using data parallel constructs available in our system using as an example the Series benchmark from Tamarin's jsbench performance suite [23], computing a series of Fourier coefficients. Even though data parallel computations in our system are expressed in ASDP and not in "full" ActionScript, due to close similarity between these two languages, the data parallel version of the Series benchmark is almost identical to the sequential one. The difference between the two versions includes replacement of the original version's sequential loop responsible for performing a trapezoid integration (Figure 1) with a definition (Figure 3) and an invocation (Figure 2) of a data parallel function implementing the same functionality. The total of 4 lines of code have been removed and 11 added, with over 200 left intact.

In order to facilitate understanding of the example, we briefly describe our support for data parallelism below and and then expand it in Section 3.

[2] Clearly, the number of modifications vary depending on the specifics of a given program as, for example, not all data types available in ActionScript are supported in ASDP.

2.1 Data parallelism support overview

The style of data parallelism supported in our system is similar to that defined by OpenCL [7] or CUDA [18]. Parallel computation is defined by a special data parallel function called *kernel*, but the programming model is simplified to better suit the requirements of a scripting language setting, as OpenCL's and, arguably to a lesser extent, CUDA's formulations are quite low level and thus somewhat difficult to use. In particular, when compared to OpenCL, we relieve the programmer from the burden of explicitly setting up devices for kernel execution, managing command queues and creating buffer objects for kernel argument passing – all of these tasks are automatically handled by our runtime.

In our system kernel definitions are similar to ActionScript functions definitions – kernels are distinguished by a "dot" symbol preceding their name (eg. the `.TrapezoidIntegrateDP` kernel definition in Figure 3). The kernel's code is executed in parallel for each point in a single-dimensional *index space* – conceptually it is equivalent to executing the kernel's code inside of a parallel loop with the number of iterations equal to the size of the index space. A given point in the index space is identified by a built-in variable `index` of type `int`, ranging from 0 to the size of the index space. We say that a kernel is *evaluated* after its code is executed for all points in the index-space. Even though kernel return type in Figure 3 is defined as `float`, the actual result of the kernel's evaluation is a vector of float values, with each separate float value representing the result of the computation at a given point in the index space.

The kernel example presented in Figure 3 uses only variables and passed-by-value arguments of types that are also available in ActionScript – we use an upcoming version of the language that features the `float` type with a standard IEEE 754 [12] semantics. Please note, that the kernel utilizes the unmodified original function used to perform trapezoid integration, which significantly contributes to the ease of the parallelization effort. The reason for introducing the special case for index 0 in the first line of the kernel is that the first element of the result is computed outside of the original loop – we simply assign it here to the appropriate element of the kernel's output.

The invocation of the `.TrapezoidIntegrateDP` kernel from the ActionScript level is presented in Figure 2. It strongly resembles a regular function call, but takes two special additional arguments specified between the square brackets – the first describes the size of the index space and the second specifies a vector of float values to be used to store the output of the kernel's evaluation (`TestArray` is defined in the original version of the benchmark as a vector of vectors of float values).

3. Language

Data parallel computations supported in our system are expressed in a Domain Specific Language (DSL) we call ASDP (ActionScript Data Parallel). The description of ASDP and of the extensions required to embed ASDP kernels into ActionScript programs, building on the overview presented in Section 2.1, is presented below.

3.1 ASDP

The basic units of data parallel execution in our system, that is kernels, are written in ASDP, which has been designed to closely resemble ActionScript, making embedding ASDP kernels into ActionScript source code feel natural.

In order to enable efficient execution of ASDP kernels on GPUs, ASDP removes certain features of ActionScript, while trying to preserve the overall ActionScript look and feel. ASDP supports ActionScript-style control-flow constructs (such as loops or conditionals), local variable declarations, function invocations and equivalents of ActionScript primitive types with the exception of `Number` (ie. `uint`, `int`, `Boolean`, `float`, `float4`). On the other hand, ASDP does not support dynamic memory allocation, recursion (both direct and indirect) [3], objects, closures or global variables. Despite these restrictions, as demonstrated by successful parallelization of our benchmarks, ASDP is fully capable of expressing realistic workloads. Similarly to other data parallel languages such as OpenCL or CUDA, ASDP also includes support for additional vector types that allow programmers to take direct advantage of vector processing support often available on modern architectures. The following types are available in ASDP :

- primitive types: `char` and `uchar` (signed and unsigned 8-bit integer), `short` and `ushort` (signed and unsigned 16-bit integer, `int` and `uint` (signed and unsigned 32-bit integer), `float` (single-precision floating point number)
- vector types: vectors of 2, 4, 8, or 16 values of every supported primitive type – the name of each such type consists of the name of the primitive type value followed by the number of values in a given vector, for example: `char2`, `float4` or `ushort8` or `int16`, etc.
- array types: arrays containing either primitive values or vector values – the name of each such type consists of the name of the primitive or vector type value followed by the word "Array", for example: `uintArray`, `char2Array`, `float4Array`, etc.

The return value of a kernel, that is a value resulting from executing a kernel's code for a single point in the index space, can be of any type supported by ASDP. These separate values are assembled together into a single data structure containing the result of the entire kernel evaluation and then made available to the ActionScript code as described in Section 3.2.

Please also note that kernels can directly call previously defined ActionScript functions, as long as the ActionScript functions respect ASDP restrictions described earlier in this section (no dynamic memory allocation, no recursion, etc.). As an example, consider the `.TrapezoidIntegrateDP` ASDP kernel function in Figure 3 calling the `.TrapezoidIntegrate` ActionScript function defined in the original sequential version of the program performing trapezoid integration.

[3] This restriction is also imposed by OpenCL and older versions of CUDA due to limitations of GPU hardware.

3.2 ActionScript extensions

Scheduling execution of ASDP kernels from the level of ActionScript resembles very strongly invocation of standard ActionScript functions. In the simplest form, one must specify only one additional "special" parameter, the size of the index space, which is specified between square brackets preceding the argument list. Let us consider a very simple kernel, `.init_kernel`, as an example:

```
function .init_kernel():int {
  return 42;
}
```

The following invocation of `.init_kernel` schedules execution of the kernel's code for every point in 1000-element index space – its evaluation results in a byte array allocated internally by the runtime containing 1000 integers, each equal to the value 42:

```
var output:ByteArray = .init_kernel[1000]();
```

The reason that the kernel evaluation result is a byte array is that `ByteArray` is the only ActionScript-level data type that is capable of encoding all ASDP vector and array types described in Section 3.1. Clearly, this can be suboptimal as further data transformations can be required at the ActionScript level, for example to transform a byte array to a "regular" ActionScript array. For that reason, to at least partially ease the programming burden due to possible subsequent data transformations, for the kernel return types that have their direct ActionScript equivalents (ie. `float`, `float4`, `int` and `uint`) we also support kernel evaluation yielding a result that is of ActionScript `Vector` type. This functionality is supported by explicitly passing the output vector as the second "special" parameter:

```
var output:Vector.<int> = new Vector.<int>(1000);
output = .init_kernel[1000, output]();
```

Similarly to the previous example, the `.init_kernel` invocation schedules execution of the kernel's code for every point in 1000-element index space, but this time its evaluation results in a 1000-element ActionScript Vector of integers, each equal to the value 42.

In order to understand how parameters are passed to kernels, let us consider another simple kernel, `.sum_kernel`, that takes two arguments of type `intArray`, which have no equivalents at the ActionScript level:

```
function .sum_kernel(in1:intArray,
                     in2:intArray ):int {
  return in1[index] + in2[index];
}
```

We handle parameter passing similarly to how we handle kernel evaluation results – at the ActionScript level the input parameters are represented by appropriately populated byte arrays. When invoking `.sum_kernel`, at the ActionScript level the arguments will be represented by two byte arrays containing the number of integer values equal to the size of the index space specified at the kernel's invocation:

```
var in1:ByteArray = new ByteArray();
var in2:ByteArray = new ByteArray();
for (var i:uint = 0; i < 1000; i++) {
  in1.writeInt(42); in2.writeInt(42);
}
output:ByteArray = .sum_kernel[1000](in1, in2);
```

This invocation yields a byte array containing 1000 integers, each equal to the value 84.

4. Implementation

Our solution is fully integrated into the existing ActionScript tool chain. We modified the upcoming ActionScript Falcon compiler [19], soon to be open-sourced, to translate programs containing ASDP kernels to *abcFiles* which store ActionScript Byte Code (*ABC*) [1], and are loaded and executed by the ActionScript virtual machine. We also modified the open-source production ActionScript virtual machine, Tamarin, to support execution of ASDP kernels and to dynamically choose the optimal processing. Consequently, programmers can keep using tools already familiar to them, which should have a great positive effect on their productivity.

4.1 Compilation

As the ASDP language closely resembles ActionScript, only moderate changes to the compiler's code have been necessary. We modified the existing ActionScript parser to support kernel definitions (special "dot" symbol preceding the name of the kernel) and their invocations (specification of "special" parameters described in Section 3.2).

Falcon compiles an ASDP kernel into two different formats: OpenCL (so that it can be executed by either a CPU OpenCL driver or a GPU OpenCL driver) and ABC (so that it can be executed sequentially such as any other ActionScript function).

4.1.1 OpenCL

From one point of view, ASDP can be considered as a variant of OpenCL language lifted to a higher level of abstraction – it omits many of the low-level aspects of OpenCL language, such as image types, address space qualifiers, or attributes. Consequently, compilation from ASDP to OpenCL language is rather straightforward, as all ASDP types have their direct OpenCL equivalents (eg. `char`, `int2`, `float16`), possibly with different names (eg. ASDP's `intArray` is OpenCL's array). ASDP's built in variable `index`, described in Section 2.1, is translated into OpenCL's `get_global_id(0)`, which is used to identify a position in the single-dimensional index space[4] during OpenCL execution. Finally, evaluation of an OpenCL kernel, similarly to an evaluation of an ASDP kernel, returns a set of values, each representing the result of the kernel's execution at a given point in the index space, which makes translation of the evaluation results straightforward.

The compiler has been modified to express translated OpenCL code as a string, so that the virtual machine can pass it directly to an OpenCL driver for execution. In order to keep ABC format modifications to the minimum, we embed the strings representing OpenCL kernels into the ABC's constant pool which is directly referred to from the modified ABC's `MethodBodyInfo` descriptor, and can be easily found during ABC parsing by the virtual machine.

4.1.2 ABC

From another point of view, ASDP is a language similar to "standard" ActionScript . Consequently, translation of ASDP kernels back to ActionScript was not complicated. A kernel is translated to a "standard" ActionScript function, that is executed by the virtual machine in a loop, and takes the `index` variable as an explicit parameter. The ASDP array and vector types (see Section 3.1) that do not exist in ActionScript are translated to ActionScript's `ByteArray` and to appropriate ActionScript's `Vector` types, respectively, with all array and vector operations translated accordingly. Unfortunately, at this point we cannot completely correctly translate all ASDP primitive types to their ActionScript equivalents, as ActionScript support for integer types is somewhat non-standard (eg. a result of an operation on two `int`s can overflow to a `Number`), but we have not experienced any problems related to this fact when translating our sample applications. The ActionScript function compiled from a given kernel retains the kernel's type. As a result, the final kernel evaluation result in this case must be assembled by the virtual machine from individual function executions inside the loop.

[4] At this point we only support single-dimensional index space as it allows programmers to express a range of algorithms while remaining simple to reason about, but there are no technical obstacles to supporting two- or three-dimensional index spaces.

4.2 Adaptivity

Tamarin can choose to execute a given kernel in three different modes: sequentially using the kernel's code translated back to ActionScript , in parallel using the OpenCL CPU driver and in parallel using the OpenCL GPU driver.

The first time a given kernel is invoked for a given index space size, no information about its previous runs is available. Consequently, during the first three executions, Tamarin schedules execution of the kernel in each of the three different modes to gather initial profiling data.

Based on the data gathered during the first three runs, the optimal mode (the one that executed the kernel fastest) is chosen to evaluate the kernel in the future, but all modes are periodically sampled to detect changes in the execution environment. Tamarin takes into consideration that effectiveness of non-optimal modes may be orders of magnitude worse than that of the optimal mode, and chooses sample ratios for non-optimal modes such that the total overhead compared with the execution in the optimal mode does not cross a certain threshold. There is a trade-off between the threshold size and the time it takes the scheduler to react to execution environment changes – the smaller the threshold, the less frequently the non-optimal modes are sampled, which makes the reaction time longer. We currently use a threshold of 10%. The process of re-evaluating the optimal mode and re-calculating a sampling ratio is repeated after each sampling run.

As the execution of a given kernel progresses, the scheduler no longer must rely on just the execution time from the previous run, but can utilize historical data to compute an estimated time. Current estimated time data is stored per-kernel in a separate bucket for each index space size range. Buckets are of non-uniform capacity and grow exponentially with the index space size. The "new" estimated time is computed by combining the freshly measured execution time information (from the run that has just finished executing) with the "previous" estimated time using the following formula ($prevEstimatedTime$ is in turn an aggregate of all past execution times computed using the same formula – in our system $\alpha = 0.5$):

$$\alpha * measuredTime + (1 - \alpha) * prevEstimatedTime$$

Timing data is acquired using high resolution timers provided by the processor.

The scheduler computes the sampling ratios by calculating how many executions in the optimal mode should happen before doing a profiling run in any of the modes. This number of executions N is computed for a given execution mode after its respective sampling run computes its current estimated time (and thus current best time is known as well), using the following formula:

$$N \leftarrow max\left(\left\lceil \frac{overhead}{bestTime * (10\%/2)} \right\rceil, 1\right),$$

$$\texttt{where} \quad overhead \leftarrow estimatedTime - bestTime$$

Intuitively, the larger the overhead in the fraction's numerator, the larger the number of iterations before a given mode will be sampled again. The formula uses half of the specified overhead since there are two non-optimal modes and an equal share of the overhead

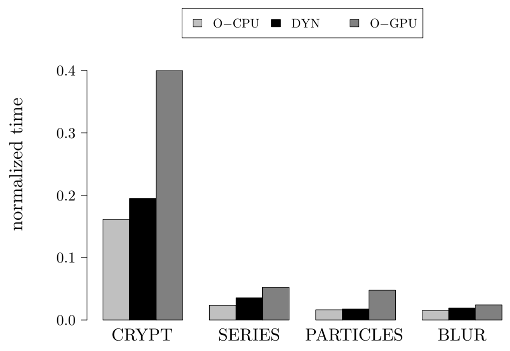

Figure 4: SMALL

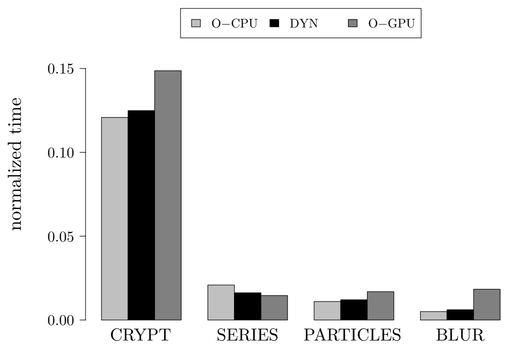

Figure 5: MEDIUM

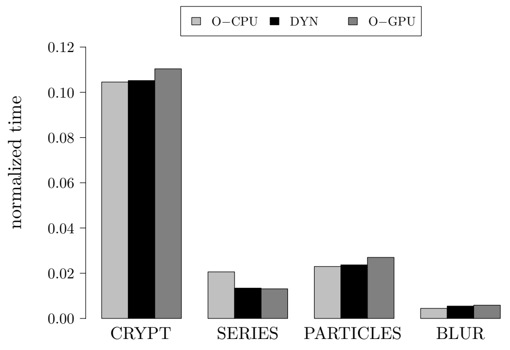

Figure 6: LARGE

Execution on DESKTOP (SMALL, MEDIUM and LARGE index space sizes)

must be assigned to each of these modes. The max operator is used to make sure at least one run occurs between each sampling run so that the computation progresses. It is straightforward to prove that using this formula we can indeed bound the total overhead over a certain number of executions to 10% (see Appendix A).

4.2.1 Initial sampling runs optimization

Since kernel evaluation times in different modes can differ by orders of magnitude, initial sampling performed during the first three kernel runs may incur an amount of overhead that will not be amortized until thousands of subsequent kernel evaluations are finished. Therefore, instead of sampling the first three whole kernel evaluations, we sample only a part of the whole kernel evaluation that occurs over a subset of the entire index space. We then extrapolate the total evaluation time for each mode, execute the rest of the kernel using extrapolated optimal mode, and use the resulting data to initialize the scheduling algorithm. Clearly, additional modifications were required to support partial kernel evaluations, but they were moderate – the main idea is to pass an additional parameter to the kernel that specifies the current offset in the kernel's index space.

5. Performance evaluation

When evaluating our system we focus on two important performance characteristics. The first is the average performance of the adaptive scheme with the runtime dynamically choosing the best execution scheme from the available options. The second is the behavior of our system under varying machine load.

We evaluated our system using four benchmarks representing different types of workloads (cryptography, physics, image processing, scientific):

1. CRYPT – IDEA encryption algorithm (from Tamarin's jsbench suite [23])
2. PARTICLES – particle simulation (adapted from the "RGB Sinks and Springs" [14] demo)
3. BLUR – blur filter (adapted from the PIXASTIC image processing library [20])
4. SERIES – a series of Fourier coefficients computation (from Tamarin's jsbench suite [23])

For each (initially sequential) benchmark a data parallel portion of the code was identified and translated into a definition and an invocation of an ASDP kernel.

In some cases the translation process was straightforward, for example in the case of the SERIES benchmark described in Section 2, in some others the need to satisfy the ASDP restrictions made it more elaborate but never very complicated and largely mechanical. For example, the original version of the "RGB Sinks and Springs" application (and thus the sequential version of the PARTICLES benchmark), the input data was encoded as ActionScript objects that are not supported in ASDP – the data parallel version has been modified to encode the input data as byte arrays.

The two versions of each benchmark (original and data parallel) enabled 5 different execution configurations:

1. ORG – sequential execution of the original benchmark

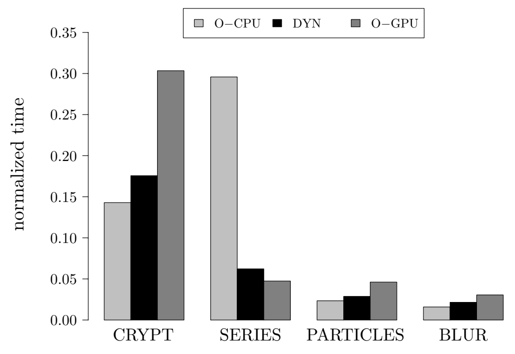

Figure 7: SMALL

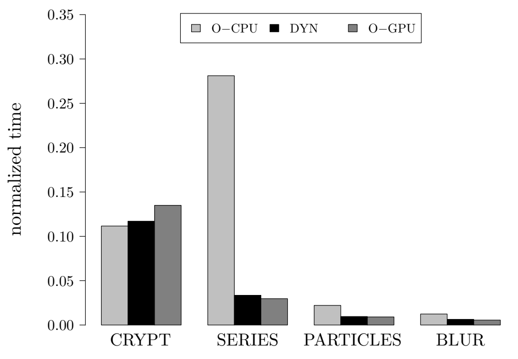

Figure 8: MEDIUM

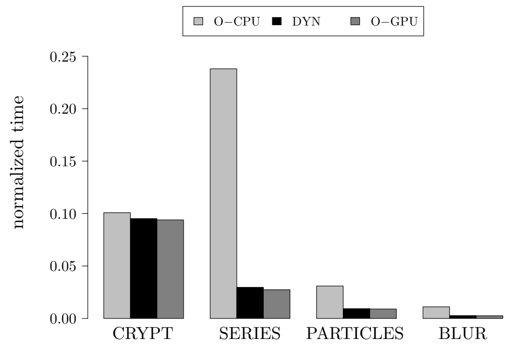

Figure 9: LARGE

Execution on LAPTOP (SMALL, MEDIUM and LARGE index space sizes)

2. SEQ – sequential execution of the parallelized benchmark (as described in Section 4, ASDP kernels are translated back to "pure" ActionScript code)
3. O-CPU – parallel execution of the parallelized benchmark utilizing OpenCL CPU driver only
4. O-GPU – parallel execution of the parallelized benchmark utilizing OpenCL GPU driver only
5. DYN – parallel execution of the parallelized benchmark where the runtime dynamically and automatically chooses the best execution mode between SEQ, O-CPU and O-GPU

As ASDP at this point does not support double-precision floating point data, all benchmarks have been modified to use only single-precision floats. Each benchmark features only a single execution of a kernel during its timed run so that during executions in the DYN configuration we can determine if the kernel was executed sequentially or in OpenCL (on either a GPU or a CPU). This required modifying the CRYPT and SERIES benchmarks to remove the second kernel invocation (along with respective portion of the computation in the original sequential version of the benchmark), which does change the result computed by the benchmarks, but does not lead to a loss of generality in terms of performance results generated.

We used two different machines to evaluate our system, each with quite dramatically different performance characteristics. The first was a 2.93GHz Intel Xeon Mac Pro (2 CPUs x 6 cores) desktop machine with a discrete AMD Radeon HD 5770 GPU card, running Mac OS X 10.6.8 in 32GB of RAM. The second was a laptop featuring AMD Fusion E-450 APU (a single 2-core 1.54GHz CPU and the AMD Radeon HD 6320 GPU integrated on on the same die), running Windows 7 Home Premium in 4GB of RAM.

5.1 Average Performance

In Figures 4-9 we plot execution times (averaged over 100 iterations within the same virtual machine invocation) for the O-CPU, O-GPU and DYN configurations, normalized with respect to the execution time of the ORG configuration [5]. Figures 4-6 plot execution times on the desktop and Figures 7-9 plot execution times on the laptop for different sizes of the index space (and thus of the output): small, medium and large – each larger by an order of magnitude from the preceding one. We omit the execution times for the SEQ configurations as, despite our initial hope that the SEQ configuration can be competitive for small kernel sizes, they were never faster than either O-CPU, O-GPU or DYN configurations.

The first conclusion that can be drawn from analyzing Figures 4-9 is that all OpenCL configurations are much faster than the execution of the original sequential benchmarks (ORG) against which they have been normalized. On the desktop, execution of the CRYPT benchmark's O-GPU configuration for the small index space is only a little over 2x faster than the original sequential execution, but the remaining desktop benchmark configurations are at least 6x faster, with all the laptop configurations being at least 3x faster than their original sequential counterparts. The relatively weak performance of the CRYPT benchmark's O-GPU configuration for the small index space on the desktop is due to small input size resulting in short computation time that cannot take full advantage of available GPU parallelism. Other than the introduction

[5] In other words, we divide execution times of the O-CPU, O-GPU and DYN configurations by the execution time of the ORG configuration.

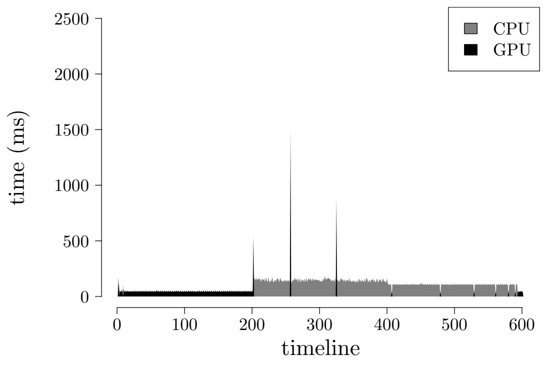

Figure 10: MEDIUM

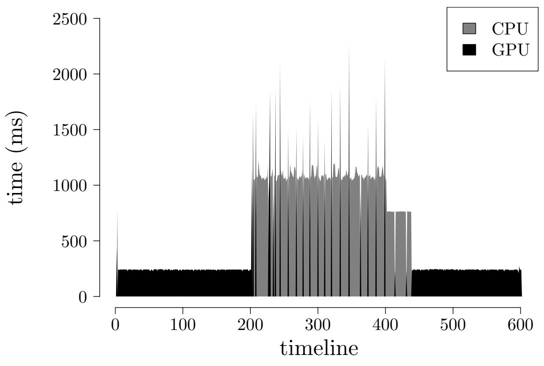

Figure 11: LARGE

PARTICLES benchmark execution– DYN configuration

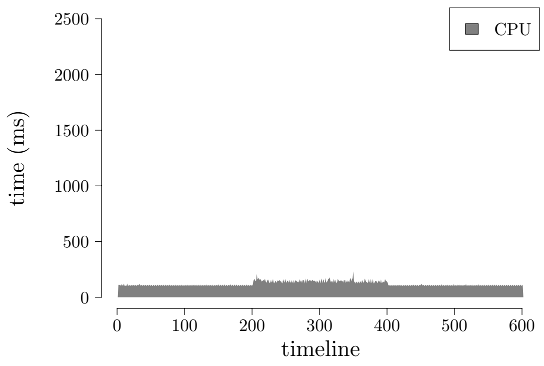

Figure 12: MEDIUM

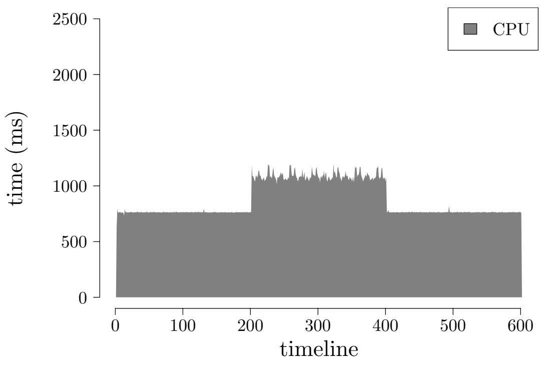

Figure 13: LARGE

PARTICLES benchmark execution – O-CPU configuration

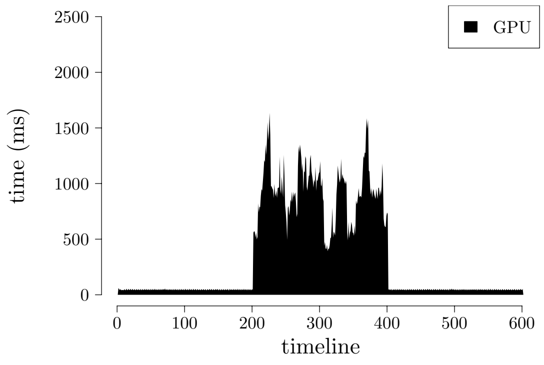

Figure 14: MEDIUM

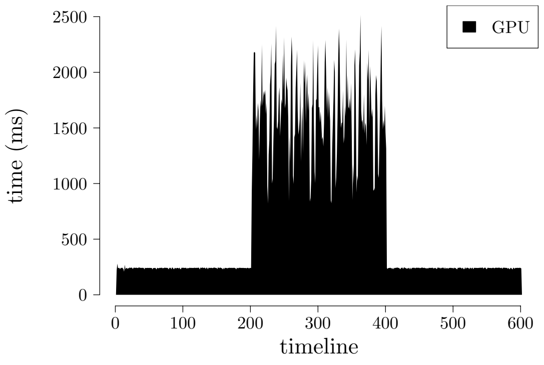

Figure 15: LARGE

PARTICLES benchmark execution – O-GPU configuration of parallelism, the main reason for the performance difference is that ASDP removes almost all dynamic features of ActionScript , which results in much faster code.

The second conclusion is that different OpenCL configurations fixed to the same processing unit behave differently on different hardware configurations. On the desktop, in most cases the O-CPU configuration is faster than the O-GPU configuration, due to the GPU card being a discrete component and high CPU-GPU communication cost. The only exception from that rule is the SERIES benchmark, as the size of its input data is very small, and the benefit of the higher degree of parallelism on a GPU outweighs the communication costs for larger sizes of the index space. On the other hand, on the laptop, in many cases the same O-CPU configuration is slower than the O-GPU configuration, as both the CPU and the GPU are integrated on the same die, which significantly reduces the communication cost. The O-CPU configuration can still be faster than the O-GPU configuration whenever the benefit of the higher degree of parallelism available on a GPU has a lower impact (eg. for smaller index space sizes).

Finally, we observe that in all cases the dynamic configuration (DYN), which automatically chooses the best processing unit to execute a given workload, closely trails the best configuration for a given machine, benchmark, and index space size. We do not always meet the 10% overhead threshold because the cost of the initial sampling run is not always amortized over the 100 kernel evaluations.

5.2 Adaptive Performance

In addition to choosing the best processing unit for a given workload on a given hardware platform in a "steady" state (where a given workload executes at least a few times to obtain initial profiling information), our system also adapts to varying machine load by switching between processing units depending on how heavily (or lightly) loaded they are at any given time during execution.

We used the PARTICLES benchmark executing on the laptop as our case study to demonstrate that our system is indeed capable [6] of this kind of behavior.

In our experimental setup the PARTICLES benchmark is executed for 600 iterations, with the GPU load changing from light to heavy and then back to light. The GPU load is created by executing the GPU Tropics benchmark [25] based on the Unigine 3D engine [24]. It is worth noting that while the Tropics demo mostly increases the load on the GPU, it also affects CPU execution by increasing its load by 10-15%.

The execution time and the processing unit[7] are recorded separately for every iteration. After the first 200 iterations the execution of the Tropics demo is triggered by the runtime, and after the following 200 iterations the execution of the demo is terminated, both actions implemented utilizing OS-level facilities available to the runtime. An additional amount of wait time is given after the execution of the demo is triggered and after it is terminated, to make sure that these actions have sufficient time to complete, but please note that this wait time has no effect on the benchmark execution times as these are measured individually.

In Figures 10-11 we plot execution times for each of the 600 iterations (the number of iterations constitutes the "timeline" for the entire execution) of the DYN configuration of the PARTICLES benchmark for the MEDIUM and LARGE sizes of the index space [8]. The color grey represents CPU executions and black GPU executions. In both cases, execution moves from the GPU to the CPU shortly after it is detected that the GPU is heavily loaded (ie. after the first 200 iterations). After the GPU load comes back to normal (ie. after the first 400 iterations), execution eventually switches back to the GPU – more quickly if the difference between CPU and GPU execution is larger (Figure 11 — right before iteration 450) and less quickly if the difference is smaller (Figure 10 – close to iteration 600). For comparison, we present equivalent graphs (same load) for the O-CPU configuration in Figures 12-13 and for the O-GPU configuration in Figures 14-15.

The conclusion is that our system is not only capable of adapting to varying machine load, but also that under a varying machine load the adaptive DYN configuration is on average faster than both the O-CPU and O-GPU configurations (respectively 102ms, 120ms, 337ms for the MEDIUM index space size and 573ms, 873ms, 708ms for the LARGE index space size).

6. Related Work

There is a large body of related work in the area of data parallel programming, including what are arguably the closest approaches to our work, that is attempts to introduce data parallelism to high level languages. Examples include CopperHead [3] (Python), Accelerator [21] (.NET), Lime (Java) [6], Aparapi [22] (Java), Accelerate [4] (Haskell), Nikola [17] (Haskell), and Data Parallel Haskell [5]. The data parallelism support provided by the RiverTrail project [13] seems to be the closest to our proposal. RiverTrail expresses data parallelism in "pure" JavaScript, but while it can be considered equivalent in terms of expressiveness to our own solution, despite also compiling data parallel constructs to OpenCL, their system currently does not support execution on a GPU [11]. WebCL [9] is another example of a solution that supports data parallel programming in JavaScript– it is a JavaScript binding to OpenCL. The WebCL's programming model is then identical to that of OpenCL's and thus equally low level. The main difference between all the solutions mentioned above and our system is that, to the best of our knowledge, none of them supports automatic selection of the optimal processing unit or adaptive processing unit switching.

Another notable piece of the related work is the Qilin system [16]. Qilin introduces an API for C++ that allows programmers to specify parallel computations and supports utilization of multiple processing units for a given workload, including adaptivity to hardware changes. However, Qilin utilizes source-to-source translation and requires manual training runs which make its practical application questionable. Furthermore, Qilin's scheduler is incapable of reacting to machine load changes, but only to physical swapping of hardware components.

Lee et al. [15] present a framework for implementing a multitude of DSLs targeting heterogeneous platforms. In comparison, our approach emphasizes a very close relationship between the host language and the DSL, where we see the highest chance of increasing programmer comfort and adoption. Furthermore, our execution scheduler is reactive, sampling-based whereas theirs is predictive, based on platform parameters.

Finally, Binotto et al. [2] describe a system where OpenCL is used as the primary programming model, and which does support automatic co-scheduling between a CPU and a GPU. Their approach, however, targets mainly scientific computing, at least partially due to the low-level nature of the OpenCL model, focuses on utilization of the entire platform rather than selection of the opti-

[6] Some benchmarks never switch their execution modes, especially those executed on the desktop, but some others (eg. BLUR) exhibit very similar dynamic behavior.

[7] In the first iteration it is actually the fastest execution unit that is reported as this processing unit is chosen to finalize execution of the first optimized profiling run, as described in Section 4.2.1.

[8] We omit the plot for the SMALL size of the index space, as the PARTICLES benchmark on the laptop executes faster on the CPU for this index space size in the first place (see Figure 7) and increasing the GPU load has no chance of triggering any processing unit switch.

$$(N-1) * bestTime + estimatedTime < N * bestTime + N * bestTime * (10\%/2)$$

Figure 16: Equation used to derive number of executions in optimal mode

$$totalOverhead = \frac{\frac{P}{N_2}t2 + \frac{P}{N_3}t3 - \frac{P}{N_2}bestTime - \frac{P}{N_3}bestTime}{P * bestTime} =$$

$$\frac{\frac{P}{N_2}(t2 - bestTime) + \frac{P}{N_3}(t3 - bestTime)}{P * bestTime} = \frac{\frac{P}{N_2}O2 + \frac{P}{N_3}O3}{P * bestTime} = \frac{\frac{O2}{N_2} + \frac{O3}{N_3}}{bestTime} =$$

$$\frac{\frac{O2}{max(\lceil \frac{O2}{bound} \rceil, 1)} + \frac{O3}{max(\lceil \frac{O3}{bound} \rceil, 1)}}{bestTime} =$$

$$= \frac{\frac{O2}{\lceil \frac{O2}{bound} \rceil} + \frac{O3}{\lceil \frac{O3}{bound} \rceil}}{bestTime} \leq \frac{\frac{O2}{\frac{O2}{bound}} + \frac{O3}{\frac{O3}{bound}}}{bestTime} = \frac{2bound}{bestTime} = \frac{2 * bestTime * 10\%}{2 * bestTime} = 10\%$$

Figure 17: Derivation demonstrating that sampling overhead is bounded

mal processing unit, and does not support dynamic adaptation to varying machine load.

7. Conclusions

In this paper we have presented a prototype of a system that utilizes a new Domain Specific Language, ASDP (ActionScript Data Parallel) to introduce data parallelism in the context of ActionScript. Programs including ASDP kernels are compiled and executed using the standard ActionScript tool chain, with the Falcon compiler and the Tamarin virtual machine being appropriately modified. We have also shown that some programs can be easily modified to use ASDP's data-parallel constructs (see Section 2.1) and that ActionScript programs parallelized using ASDP can be multiple times faster than their sequential counterparts.

A. Appendix

The formula used to compute how many executions in the optimal mode should happen before doing a profiling run in any given mode is derived from the equation presented in Figure 16, where $estimatedTime$ is the execution time for a given mode predicted based on the current execution time measurement combined with "historical" data for this particular mode, and $bestTime$ is the shortest of all estimated times representing execution in the optimal mode (for details of the scheduler description see Section 4.2). Intuitively, the time to execute $N-1$ runs in optimal mode plus the time to execute one run in the non-optimal mode should be smaller than N executions in optimal mode plus certain overhead (in this case – 10%). As mentioned in Section 4.2, the formula uses half the specified overhead since there are two non-optimal modes and an equal share of the overhead must be assigned to each of them. This equation is easily converted into the following form, which directly represents the formula presented in Section 4.2 (the formula chooses the smallest such N):

$$N > \frac{estimatedTime - bestTime}{bestTime * (10\%/2)}$$

Let t_i (for $i = 1 : 3$) represent the current estimated execution times for each execution mode. Let us assume, without loss of generality, that $min(t1, t2, t3) == t1 == bestTime$. Let O_i $(= t_i - bestTime)$ represent the execution overhead for each execution mode. We can now express the number of executions that should happen before doing a profiling run in mode i, by transforming the formula from Section 4.2 to the following form:

$$N_i \leftarrow max\left(\left\lceil \frac{O_i}{bound} \right\rceil, 1\right),$$

$$\texttt{where} \;\; bound \leftarrow bestTime * (10\%/2)$$

We will now show that the total overhead compared to the execution in optimal mode, for a certain period P is indeed no larger than 10%.

Let us choose period $P = N_1 * N_2 * N_3$. The total execution time over this period ($totalTime$) is equal to the time spent in sampling runs plus the time spent executing the remaining runs in the optimal mode (the number of sampling runs for a given mode is equal to the length of period P divided by the number of optimal runs between each sampling run for a given mode):

$$\frac{P}{N_1}t1 + \frac{P}{N_2}t2 + \frac{P}{N_3}t3 + \left(P - \frac{P}{N_1} + \frac{P}{N_2} + \frac{P}{N_3}\right) bestTime$$

Let $optimalTime$ $(= P * bestTime)$ represent the time to execute P runs in the optimal mode. Then, the total overhead can be expressed as follows:

$$totalOverhead = \frac{totalTime - optimalTime}{optimalTime}$$

Considering that $t1 == bestTime$ and assuming that $O_2 > 0$ and $O_3 > 0$, derivation presented in Figure 17 shows that sampling overhead is indeed no larger than 10%.

References

[1] Adobe. The ActionScript byte code (ABC) format. `http://learn.adobe.com/wiki/display/AVM2/4.+The+ActionScript+Byte+Code+(abc)+format`.

[2] Alecio P. D. Binotto, Carlos E. Pereira, Arjan Kuijper, Andre Stork, and Dieter W. Fellner. An effective dynamic scheduling runtime and tuning system for heterogeneous multi and many-core desktop platforms. In *HPCC*, 2011.

[3] Bryan C. Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *PPoPP*, 2011.

[4] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *DAMP*, 2011.

[5] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *DAMP*, 2007.

[6] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen Fink. Compiling a high-level language for gpus (via language support for architectures and compilers). In *PLDI*, 2012.

[7] Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`, 2012.

[8] Khronos Group. OpenGL - the industry's foundation for high performance graphics. `http://www.opengl.org/`, 2012.

[9] Khronos Group. WebCL - heterogeneous parallel computing in HTML5 web browsers. `http://www.khronos.org/webcl/`, 2012.

[10] Web Hypertext Application Technology Working Group. Web workers draft recommendation. `http://www.whatwg.org/specs/web-apps/current-work/complete/workers.html`, 2011.

[11] Stephan Herhut. Building a computing highway for web applications. `http://blogs.intel.com/research/2011/09/15/pjs/`, 2011.

[12] IEEE. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, 2008.

[13] Intel. River trail: Bringing parallelism to web applications on top of intel opencl sdk. `http://software.intel.com/en-us/articles/opencl-river-trail/`, 2011.

[14] Barbara Kaskosz and Dan Gries. RGB sinks and springs. `http://www.flashandmath.com/advanced/rgbsinks/`.

[15] HyoukJoong Lee, Kevin Brown, Arvind Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 2011.

[16] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO*, 2009.

[17] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled gpu functions in haskell. In *Haskell Symposium*, 2010.

[18] NVIDIA. CUDA – parallel programming made easy. `http://www.nvidia.com/object/cuda_home_new.html`, 2012.

[19] Michael Schmalle. Apacheflex :: Falcon compiler update. `http://blog.teotigraphix.com/2012/01/19/apacheflex-falcon-compiler-update/`, 2012.

[20] Jacob Seidelin. PIXASTIC – JavaScript image processing library. `http://www.pixastic.com/`.

[21] Satnam Singh. Declarative data-parallel programming with the accelerator system. In *DAMP*, 2010.

[22] Aparapi team. Aparapi. `http://code.google.com/p/aparapi/`.

[23] Tamarin VM team. jsbench. `http://hg.mozilla.org/tamarin-redux/file/9e56a8e5b17c/test/performance/jsbench`.

[24] Unigine. Unigine – engine of virtual worlds. `http://unigine.com/`.

[25] Unigine. Tropics benchmark 1.3. `http://unigine.com/products/tropics/`, 2010.